CS402
Introduction to Logic in Computer Science

Coursework 3: z3 SMT Solver & Basic Program Verification
Due on 18:00, 7 June 2016

1 Getting familiar with z3 SMT Solver

Solve the following puzzles using `z3` SMT Solver (`https://github.com/Z3Prover/z3`).

> You have three crates, one has only apples, one has only oranges, and one has both apples and oranges. However, someone has maliciously relabeled the crates, and all you know is that all the labels have been shuffled, i.e., none of the crates has the right label. The goal is to infer the right label by picking a single fruit from one of the crates.

1. Suppose you picked a fruit out of the crate labeled "Mixed", and it happened to be an apple. Write a Z3 formula involving the variables `realApple`, `realOrange` and `realMixed`, such that the variable `realOrange` will tell you the real contents of the crate labeled "orange", the variable `realApple` will tell you the real contents of the crate labeled "apple", and similarly for `realMixed`. The values of these three variables should be consistent with the fact that you picked an apple from the mixed crate.
2. Use the solver to prove that there is only one possible assignment for the labels given the information that you picked an apple from the mixed crate.
3. If you had picked an apple from the crate labeled "orange" instead, would that information had been enough to uniquely determine the solution? Provide a formula that can answer this question.

Hint: consult the enumeration type (Scalas) in the `z3` SMT Solver tutorial (`http://rise4fun.com/z3/tutorial`).

2 Program Verification

`soot`(`https://github.com/Sable/soot`) is a Java program analysis framework that can translate Java code into the Static Single Assignment format (called `Jimple` in `soot`). The coursework files will contain an example script that will do the translation. Your task is to write a simple program verification tool, using `soot` and `z3` SMT Solver, based on the following assumptions:

- We will only verify Java source code (`.java` files) that contains a single public class (called `TestMe`), whici in turn contains a single public method with the signature of `public int testMe(int x1, int x2, ...)`.
- The method `testMe` will only take arguments of type `int`, and have only local variables of type `int`.
- The method `testMe` will only contain either variable declarations or assignments. Both can take an arbitrary expression, using basic arithmetic operators (`+`, `-`, `*`, `/`).
- The last line of the method `testMe` will be an assertion written in a single binary comparison operator (for example, `assert(z > x1 + 3 - x3);`.

1. **First task**: Given the assumptions, write a program that verifies whether the assertion is valid.

2. **Second task**: Now assume that the method `testMe` can have nested `if` branch statements, predicates of which are Boolean expressions. Also, the final assertion can now be a Boolean *expression* (not single comparison) that includes logical operators such as `&&`, `||`, and `!`. For example, branch predicates such as `x > 3 || y * 2 < 14` and assertions such as `assert((x > 3)|| (y < z + 3 && z > x))` are both possible. Extend your verifier to handle the branching and the more sophisticated assertions.

If the assertion is valid, the program should print `VALID`; otherwise, print out the values of the method arguments which will violate the assertion.

Important:

- `soot` currently only works with Java 7. Please adjust your development environment accordingly.
- Essentially, your program should take a single argument from the Command Line Interface (CLI) (for example, `$ python verifier.py Test.java`), which is a `.java` file.
- Subsequently, it should use `soot` to generate the `.jimple` file, read it, then construct the `z3` SMT Solver formulas that are required for the verification. Finally, your program should invoke `z3` SMT Solver using the formulas as the input, and report the output of `z3` SMT Solver.
- By default, `soot` will enclose the assertion processing in a branch, whose predicate specifies whether the assertion should be checked at all (this will appear as `<Test: boolean $assertionsDisabled>`). You can safely ignore this.

3 Deliverables

Each person should submit the following deliverables by the submission deadline:

- **Implementation**: code for two problems, self-contained in separate directories (see below).
- **Report**: include a written report that contains detailed descriptions of how you approached the problems. For the Nonogram solver, clearly outline how you encoded the puzzle into SAT. There is no page limit.

For ease of marking, follow the following directory structure, and submit a zip file containing the top level directory, through KLMS.

```
[your student number]
├── report.pdf ..... Your report documenting both implementations
├── puzzle ..... Three .z3 files, each with formulas that answer the puzzle
└── verifier ..... Program verifier
    └── soot ..... the soot directory distributed in the coursework file
```

4 Guidelines

- You are free to choose any programming language.
- **BUT we expect solutions for both tasks to run straight out of the box.** Your submission should be self-contained, and it should run on Unix-like systems (apologies to Windows users, but `minisat` depends on `cygwin` on Windows, which makes it *harder than NP-hard* to mark these things). You can assume that `minisat` is available on path.
- Avoid printing anything other than the final solution, as it will make marking harder.
- Plagiarism will only result in zero marks; do your own work.
- Document your implementation as best as you can; whatever you do not describe, you risk having it being overlooked by the markers!

5 Evaluation and Competition

The following marking criteria apply:

- Solutions of the puzzle: 15%
- Can verify programs without branches correctly: 30%
- Can verity programs with branches : 40%
- Quality of code and documentation: 15%

Submissions will be marked using private inputs as well as the public test cases that are included in the coursework file.